Assignment 1, Part - 2

Benchmarking Lua's GC

By Sourodeep Datta, 21CS10064

1 Instruction Execution Percentages

For calculating the instructions executed by the GC, as I had mentioned in my report for Part A, I will use the number of instructions executed by $luaC\_step$, which is function that performs the garbage collection step and lua_gc which does GC initialization, setup and performs the full GC collection cycle.

1.1. Default Settings

FullGC

Total Instructions Executed = 948,626,884
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 294,369,265 + 938,989
= 295,308,254
Percentage Instructions Executed by GC = 31.13%

IncrementalGC

Total Instructions Executed = 1,162,738,434
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 79 + 249,665,490
= 249,665,569
Percentage Instructions Executed by GC = 21.47%

GenerationalGC

Total Instructions Executed = 1,067,980,600
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 15,636 + 199,775,187
= 199,790,823
Percentage Instructions Executed by GC = 18.71%

1.2. m = 100, n = 100

FullGC

Total Instructions Executed = 95,583,146
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 20,453,875 + 118,075
= 20,571,950
Percentage Instructions Executed by GC = 21.52%

IncrementalGC

Total Instructions Executed = 105,532,940
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 79 + 19,722,905
= 19,722,984
Percentage Instructions Executed by GC = 18.69%

GenerationalGC

Total Instructions Executed = 106,220,585
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 15,636 + 16,149,224
= 16,164,860
Percentage Instructions Executed by GC = 15.22%

1.3. m = 500, n = 100

FullGC

Total Instructions Executed = 474,711,903
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 102,193,765 + 482,963
= 102,676,728
Percentage Instructions Executed by GC = 21.63%

IncrementalGC

Total Instructions Executed = 582,715,043
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 79 + 107,578,304
= 107,578,383
Percentage Instructions Executed by GC = 18.46%

GenerationalGC

Total Instructions Executed = 542,613,139
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 15,636 + 86,310,494
= 86,326,130
Percentage Instructions Executed by GC (Including Setup) = 15.91%

1.4. m = 5000, n = 100

FullGC

Total Instructions Executed = 5,267,537,985
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 1,549,298,361 + 4,587,104
= 1,553,885,465
Percentage Instructions Executed by GC = 29.50%

IncrementalGC

Total Instructions Executed = 5,711,090,159
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 79 + 880,368,312
= 880,368,391
Percentage Instructions Executed by GC = 15.42%

GenerationalGC

Total Instructions Executed = 5,578,265,440
Instructions Executed by GC = $I_{lua\_gc} + I_{luaC\_step}$
= 15,637 + 1,070,019,512
= 1,070,035,149
Percentage Instructions Executed by GC = 19.18%

1.5. Analysis

When comparing between **FullGC**, **IncrementalGC** and **GenerationalGC**, it is clear that FullGC results in less instructions being executed as compared to the other 2, while having the most percentage of the instructions being related to GC functions. This might be because of additional overheads introduced by IncrementalGC and GenerationalGC that are not directly measured by the instructions executed by their functions. These can be write barriers and temporary memory allocations and their freeing.

Next, we'll look at the change in these values as the matrix size increases:
Note increasing order of matrix sizes is: [(100,100), (500,100), default: (1000,100), (5000,100)]

1. We see that the percentage of instructions executed by FullGC increases. This may be due to significantly more objects being created as the matrix size increases, thus resulting in a much long traversal having to be done at the end when the collection cycle is done (as no GC had been done during the script execution).

2. As the matrix size increases, an inversion occurs in the percentage execution of instructions by the incremental and generational GCs. This can be attributed to the fact that the IncrementalGC mostly keeps up with the increase in size of the matrix, maintaining roughly the same percentage of executed instructions, while the generational GC's increases. This could be due to the number of objects being created increasing significantly as the matrix size increases. The incremental GC can handled it by dynamically increasing how long it waits before the next step of the collection, in order to have to do less steps. This is not done by the Generational GC which does a collection every time a certain threshold is passed. This can be seen in the call graphs, where the calls made to *luaC_step* is much less in IncrementalGC compared to GenerationalGC.

2 Perf Analysis

1. Branch Misses: It is relevant as it gives us an idea of the complexity of the control flow

Table 1: Performance Analysis Metrics for Different GC Configurations

Metric	**Full GC**	**Incremental GC**	**Generational GC**	**Without GC**
Branch Misses	2,440,799	2,117,569	2,277,231	1,084,735
Page Faults	32,437	16,517	14,886	16,263
Cache Misses	582,772	12,195,867	13,495,368	239,520
Instructions Per Cycle	2.14	1.23	1.08	2.47

i.e. a measure of the conditional statements that the GC has to execute, and the degree to which branches can be predicted.

Full GC has the most branch misses due to having to go through all the objects allocated, which may lead to a complex control flow.
Without GC has the least as it doesn't need to do the garbage collection cycle.

2. Page Faults: It is relevant as GC processes may access large amounts of memory, possibly leading to page faults, especially if memory is heavily fragmented or if large amounts of memory are being managed. It gives information about the fragmentation of the memory.

   Full GC has the most page faults because it has to go through all the objects that have been allocated in memory. Many of the objects allocated earlier may be in pages that are no longer mapped to physical RAM, causing a page fault.
   Generational GC has slightly lower page faults than Incremental GC because objects in the younger generation are more likely to still be mapped to physical memory.
   Without GC has more page faults than generational GC. This may be due to memory fragmentation. The generational GC compacts the memory after freeing dead objects, which might lead to overall lower number page faults. Without GC does not do that, so it need to allocate new pages each time all the memory of the previous page has been allocated.

3. Cache Misses: It is relevant as it gives an idea about the locality of reference of memory accesses of each program, better than page faults (due to the much lower cache size).

   Full GC has the lower cache misses compared to Incremental and Generational GC due to running only at the end, after the script has finished creating the matrix. The other 2 GCs run while the matrix is being created, thus frequently replacing data in the cache with data needed by the GC while executing it's collection step.
   Without GC has the least cache misses due to no GC running, thus only misses are due to the matrix creation script.

4. Instructions Per Cycle: It is relevant as it gives an overall metric to measures the efficiency of the CPU's execution of instructions.

   Full GC has the highest IPC among the 3 GC setups. This is expected as it only runs the collection at the end and has the lowest cache misses, meaning data is usually immediately accessible when required by an instruction. In fact the benefit of not having to interleave the GC collection cycle with the testbench script is quite clear, with Full GC having a 90% higher IPC than the next best, Incremental GC.

Without GC has the highest IPC due to not having to run the GC during program execution, thus causing minimal cache misses and not having to ever stop the world.